International Journal of Serious Games

ISSN: 2384-8766
https://journal.seriousgamessociety.org/

Article

Using Visual Programming Games to Study Novice Programmers

Christian DeLozier[1] and James Shey[1]

[1]*Electrical and Computer Engineering Department, United States Naval Academy, Annapolis, USA*
delozier@usna.edu ; shey@usna.edu

Keywords:

Serious Games
Game-Based Learning
Visual Programming
Parallel Programming
Novice Programmers

Received: November 2022
Accepted: May 2023
Published: June 2023
DOI: 10.17083/ijsg.v10i2.577

Abstract

Enabling programmers to write correct and efficient parallel code remains an important challenge, and the prevalence of on-chip accelerators exacerbates this challenge. Novice programmers, especially those in disciplines outside of Computer Science and Computer Engineering, need to be able to write code that exploits parallelism and heterogeneity, but the frameworks for writing parallel and heterogeneous programs expect expert knowledge and experience. More effort must be put into understanding how novice programmers solve parallel problems. Unfortunately, novice programmers are difficult to study because they are, by definition, novices. We have designed a visual programming language and game-based framework for studying how novice programmers solve parallel problems. This tool was used to conduct an initial study on 95 undergraduate students with little to no prior programming experience. 71% of all volunteer participants completed the study in 48 minutes on average. This study demonstrated that novice programmers could solve parallel problems, and this framework can be used to conduct more thorough studies of how novice programmers approach parallel code

1. Introduction

Writing parallel programs remains a challenge, even for expert programmers [1], but parallel hardware is ubiquitous and must be exploited by applications to achieve the best performance. For example, general-purpose computing on graphics processing units (GPGPU) and tensor processing units (TPU) have been used to accelerate certain applications using parallelism compared to traditional processors [2]. In fast-growing domains such as Data Science, domain knowledge experts in vastly different areas such as digital signal processing, medicine, and finance, need to be heavily involved in writing code to solve problems within their domain to find optimal solutions [3]. However, domain knowledge experts cannot be expected to be both experts in their domains and experts in parallel programming. Likewise, parallel programming experts cannot be expected to fully understand the intricacies of all of these domains.

Programming languages and frameworks should enable less experienced programmers to write efficient code for applications within their domain of knowledge.

To this end, parallel programming languages must be designed such that correct and efficient programs can be written by domain experts who may be novice parallel programmers. By studying how novice programmers solve problems with parallel programming languages, we can better design parallel programming languages and compare programming language features. Unfortunately, it is difficult to study novice programmers because they are, by definition, novices and do not know how to write parallel code. As such, prior studies on how programmers understand parallel programming frameworks have focused on advanced undergraduate students and graduate students in computing majors [4, 5]. Given instruction on how to write parallel code, a programmer will likely solve parallel programming tasks in a way that corresponds with their education.

We have developed a graphical programming language and framework for studying how novice programmers solve programming problems. Graphical programming languages [6–9] allow programmers to solve problems using blocks corresponding to programming constructs. Using a graphical programming language prevents novice programmers from making syntax errors, which are typographical bugs in the way a program is written. Blocks are designed to prevent syntax errors by only combining in ways that make sense for a program. For example, the block that starts a program will only allow blocks to be attached after it instead of before it considering that it does not make sense to have blocks before the start of a program.

Alongside the graphical programming language, we co-developed a game that introduces basic parallel programming concepts and assesses how the player solves parallel programming problems. Our framework is based on prior work on teaching programming to beginners such as in the "hour of code" challenge [10]. In the "hour of code" challenge, grade-school students learn basic programming concepts and skills using visual programming languages to perform tasks in a game environment. Using this framework, we conducted a preliminary study of how novice programmers solve parallel programming problems. This study was conducted on volunteer participants who were recruited from an introductory core course on cyber security and sought to validate the framework.

The contributions of the paper are as follows:

- Proposes a novel use of visual programming languages and serious games for studying novice programmers’ ability to understand and use complex language features
- Demonstrates, with a user study, the use of this framework to evaluate novice programmers’ ability to solve parallel programming problems involving synchronization
- Provides a framework for future researchers to use to evaluate the usability of programming language features

The remainder of this paper is organized as follows. Section 2 introduces relevant background material on visual programming languages and parallel programming. Section 3 first describes the programming frameworks that we used to develop the graphical programming language and corresponding game and then describes how the parallel programming problems included in the game relate to real-world parallel programming tasks. Section 4 discusses the implementation of this framework and technical challenges. Section 5 presents the results of the study that we conducted using this framework. Section 6 suggests future improvements that can be made to this framework and concludes the paper.

2. Background

2.1 Visual Programming

Visual block-based programming is an effective way to introduce programming. It is used at all levels of introductory programming from preschool [6, 7], to high school [8], to college [9]. The wide range of ages and skill levels highlights the flexibility of visual programming. Visual block-based programming lowers the initial entry hurdles of many conventional programming languages using drag-and-drop puzzle blocks [11]. These languages appeal to novice programmers as they are often browser-based and do not require the installation of tools. Additionally, these programming languages use visual cues, such as puzzle tabs, to indicate to the programmer how blocks can be connected together. This allows blocks to only be connected in particular ways and prevents syntax errors [12]. This ease of use often hides the fact that many of these languages still support sophisticated programming constructs to include complex data structures, file handling, arrays, mouse and keyboard inputs, and parallel code [13].

Programming requires computational, algorithmic, and logical thinking [14]. Solving a problem then requires problem identification, understanding syntax, semantics, and com plexity of a programming language [14]. Visual programming languages have the ability to introduce these topics to a wide audience. These languages have been available since the 1990s with LogoBlocks, but were limited in scope and availability [15]. Over the past decade, the field of introductory programming languages has proliferated with many different visual programming languages [14]. This explosion of options and the ease of use of the languages have fostered an environment where younger children can be introduced to programming [13]. There are many options, but popular ones include code.org, Scratch, and Alice. In [6], Scratch was introduced to preschool educators and evaluated basic computational and logical thinking. All educators saw the direct benefits and 85.7% said it should be taught to preschool students. Continuing this idea, [16] concludes that the early introduction of computational and logical thinking into the kindergarten curriculum helps with cognitive thinking. Comparing the effectiveness of block-based programming to conventional text-based programming, [8] evaluated high-school students learning under different modalities. They concluded that students that used the block-based method could transfer their gained knowledge to other modalities and performed as well as or better than their conventional text-based programming classmates. Counter to this finding [17], found statistically insignificant differences in cognitive abilities of 8 and 9-year-old students after learning to program with visual and text-based languages. The benefits of visual programming languages continue to propel them into the classroom to introduce programming concepts and computational thinking [11].

2.2 Serious Games

Prior work has assessed the use of serious games to teach programming in an interactive and engaging manner, leading to improved learning outcomes [18–22]. In the context of programming education, serious games provide a fun and interactive environment for students to learn programming concepts, practice problem-solving skills, and develop algorithmic thinking. Serious games can also adapt to learners as they progress through the game [23].

Assessing learning outcomes in serious games is a crucial aspect of evaluating their effecttiveness as a teaching tool, and games may use a variety of measures to assess learning, including time on task, number of errors and correct answers, and observable behavior changes [24, 25]. Likewise, engagement can be assessed through factors such as time in the game, comments about the game, and motivation to play and learn. For our purposes, we are mostly interested in the usability of serious games due to the relation to the usability of the programming systems we are trying to evaluate with a serious game. Factors such as time spent learning how to use the system, technical error, controls, clarity of tasks, ease of use of the system, clarity of rules, clarity of effects of actions in the game, and interface are critical to this framework [26–29].

2.3 Parallel Programming

Our initial goal in designing this system was to understand how novice programmers solved parallel programming problems. In this section, we provide relevant background on parallel programming terms and concepts.

2.3.1 Parallelism and Synchronization

Computer programs run as a stream of instructions that are executed by the central processing unit (CPU). A *sequential* program runs a single stream of instructions. A parallel program runs multiple streams of instructions at the same time. These independent, parallel streams of instructions are referred to as *threads or processes*. By executing more than one instruction at a time, parallel programs can accelerate computations compared to sequential programs. For example, a bank can efficiently update account balances by processing multiple transactions at a time. Unfortunately, the increase in speed provided by parallel programs often comes at the cost of potential bugs in the program.

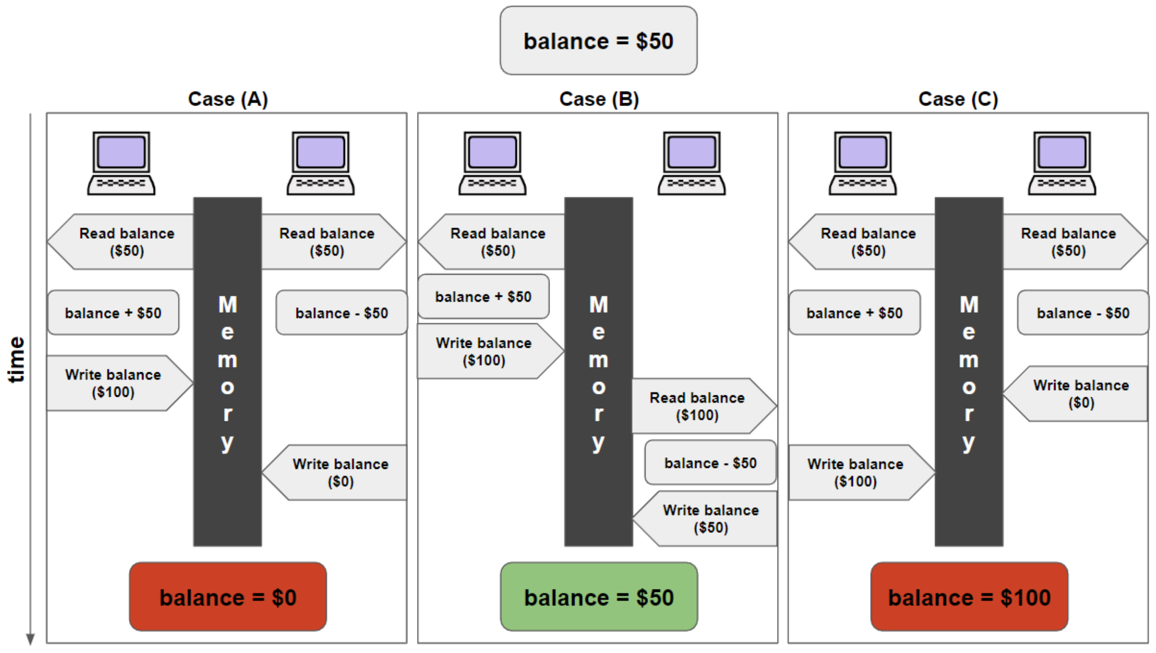


Figure 1. A demonstration of data races in a parallel program.

Program instructions can *access* data in the computer's memory. In the previous example, a bank account balance would be considered data. Access to data in the computer's memory can be a *read*, which examines the data, or a *write*, which modifies the data. A *data race* occurs when two instructions access the same data at the same time and at least one of the accesses is a write [30]. Data races can cause abnormal behavior during the execution of a parallel program because the result of a data race is generally undefined, meaning that the outcome is unpredictable. Assume that you perform a deposit to your bank account, and, at the same instant, a family member performs a withdrawal. On a modern computer, if these two transactions are processed at the same instant without any intervention, the parallel execution might result in any of the following scenarios. In the normal, case, the computer processes the transactions in the correct order, and the account balance is updated properly. However, a modern computer system will likely read the account balance in both parallel streams of execution simultaneously. Then, either the write for the withdrawal or the write for the deposit will occur first. If the deposit happens first and the withdrawal second, the computer's memory will only reflect the result of the withdrawal, leaving you with less money than you expected. If the withdrawal happens first, the computer's memory will only reflect the result of the deposit, leaving the bank with less money than expected. Figure 1 demonstrates these scenarios visually. In Cases (A) and (C), both computers read the balance at the same time. The left computer decreases the balance, and the right computer increases the balance. The final balance is determined by the order in which the computers write the balance back to memory. (Note: The write back to memory cannot occur at the same time electronically, but the reads can.) Case (B) shows the correct execution in which one of the computers completes its operation before the other reads the balance, yielding an ordering between the two operations.

Synchronization prevents data races by enforcing an order between instructions in a parallel program. By enforcing an ordering between instructions, synchronization breaks the data race condition that accesses happen at the same time. From Figure 1, correct synchronization can ensure that case (B) always happens. There are many forms of synchronization, but for this study, we focus on locks and condition variables. *Locks* support the construction of a critical section in which the memory accessed by a thread cannot be interfered with by another thread. To properly use a lock, a thread must acquire the lock, access any memory associated with the lock, and then release the lock. Any additional threads wishing to use the associated memory must attempt to acquire the same lock. A lock can only be used by a single thread at a time, thus preventing data races on the associated memory. A *condition variable* allows a thread to wait to be woken up by another thread. Condition variables are used in cases when ordering matters - a thread may need to wait for another thread to complete a task before it completes its own task.

2.3.2 Prior User Studies on the Usability of Parallel Programming Languages

Previous studies on the usability of parallel programming languages involve studying postgrad students and programmers with extensive programming experience [4, 5, 31, 32]. In [4], 69 graduate students across four courses completed parallel programming tasks as part of their classwork. Students used either OpenMP [33] or MPI [34] to write parallel programs. OpenMP is a parallel programming framework that allows programmers to parallelize code using annotations. For example, a loop can be executed in parallel by adding `#omp parallel for` before the loop. OpenMP handles the details of creating and distributing work to threads for the programmer. MPI provides an alternative form of synchronization in parallel code using message passing. With MPI, threads produce data and communicate that data to other threads via messages instead of saving data in shared memory. Student effort was measured in terms of the person hours spent and the number of lines of code required to complete the task. The prior experience of most participants in this study was limited to previous coursework. Data was collected using instrumented classroom computers that automatically recorded timestamps and performance data. [31] assessed the use of self-reported metrics versus automatically collected data in a similar context. This pair of studies observed that lines of code alone was not a sufficient metric for programmer effort, specifically when comparing different programming frameworks. [5] compared two parallel programming techniques, locks and transactional memory, in a study on six teams of graduate students in a graduate-level course. Researchers measured the amount of effort required to implement parallel programs using either locks or transactional memory using lines of code, person hours of effort, and qualitative interviews with the teams. The performance of the resulting code was also measured and compared. This study found that the teams using transactional memory spent less time debugging but more time optimizing performance. The final code from the transactional memory teams was easier to read compared to the lock-based code. In [32], a user study was conducted on upper-level undergraduate and graduate students to determine how effective these students were at debugging data races and deadlocks. A deadlock occurs when two processes hold a resource the other needs to continue and are therefore they are perpetually waiting for the other process to release the resource and ultimately the processes are unable to complete the program. Participants in this study were asked to properly synchronize a piece of parallel code twice, once with the help of output from a data race detector and once with output from a deadlock detector. Participants' responses were evaluated for correctness. Participants also self-evaluated their prior experience in writing parallel programs, and a set of knowledge check questions were used to validate their self-assessments. This study demonstrated that participants were more likely to correctly synchronize programs given a deadlock report than given a data race report. In summary, prior studies on parallel programming have been used to compare competing approaches to writing parallel programs by studying students with significant prior programming experience that may affect how they choose to solve these problems.

3. Design

To study novice programmers writing parallel programs, we needed to overcome three major challenges. First, novice programmers are, by definition, new to programming and tend to have trouble writing syntactically correct code. Second, to identify how novice programmers solve parallel programming problems, the problems used in the study needed to match closely to real-world parallel kernels. Third, teaching novice programmers about programming or parallel programming as part of the study may bias them toward solutions that have been taught, rather than coming to solutions that are more natural to them. Our experimental framework was designed to limit the prior knowledge required of novice programmers.

3.1 Graphical Programming

To help novice programmers write syntactically correct programs, we chose to implement our study using a custom graphical programming language. Programs like the "hour of code" have demonstrated that novice programmers, even those in elementary school, can solve problems using graphical programming languages [7, 8, 10]. Prior work on serious games has identified that block-based programming is a common pattern in programming games related to conditional and iterative control structures [35]. The programming language for our study was implemented using Google Blockly [36].

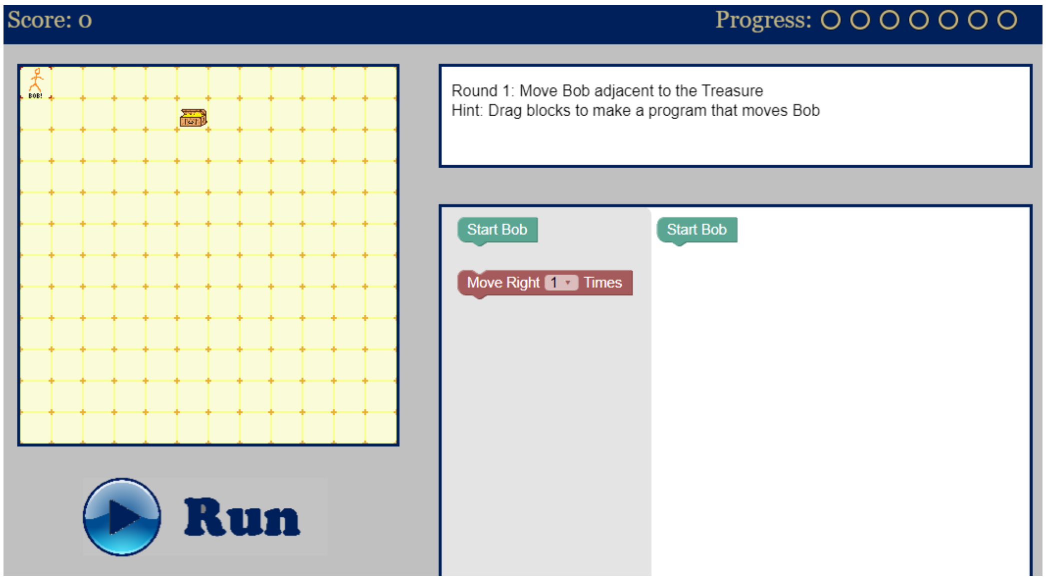

Round 1: Move Bob adjacent to the Treasure
Hint: Drag blocks to make a program that moves Bob

Figure 2. Problem 1 introduces movement with block programming and a single worker.

Figure 2 shows the development environment for programs written as part of this study. At the beginning of the study, the programmer has access to only two blocks - Start Token and Move Right. Start Token is equivalent to the start of a thread in a parallel program, and Move Right is an action that can be taken by a token on the game board, which is equivalent to an action taken by a thread in a parallel program. As the participant progresses through the study, they gain access to additional and more complex blocks. Each new block is explained to the participant by example in the introductory rounds of the study. Once the participant has completed the example problems and learned all of the required blocks, they move on to solving parallel programming problems using these blocks.

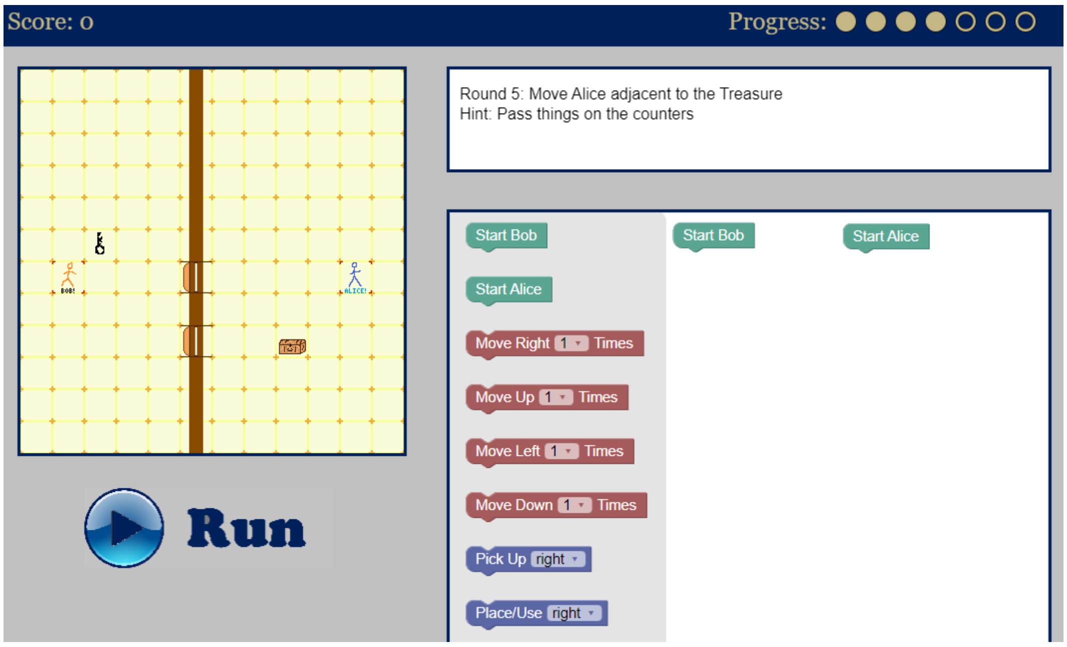
3.2 Gamification of Parallel Programming Tasks

We identified common parallel design patterns and mapped them to game scenarios in this study. Each of the levels within the game represents a parallel programming task, and the game's mechanics are designed to mimic parallel programming constructs. The two sprites, Alice and Bob, represent threads that can execute tasks. Users build a stream of instructions for Alice and Bob to execute using the visual programming language, and the game executes these streams of instructions in parallel. These levels are broken into three categories: threading or multitasking, coordination, and synchronization [37]. Threading is multiple independent operations happening concurrently. An example of threading is independent queries to a database. Coordination requires independent threads to pass information to one another. A common example is the Producer-Consumer algorithm in which one thread produces data and the other thread consumes that data. Synchronization allows threads to safely compete for a shared resource. Locks are commonly used to provide synchronization in parallel programs. An example of this is the exclusive use of a shared resource, such as a queue.

The first four levels constitute the threading category. Level one represents a single-threaded serial program. This level introduces the visual programming language and environment. Levels two and three of the game represent parallel programming tasks that are embarrassingly parallel, meaning that there are no dependencies between the parallel tasks executing on independent threads. Embarrassingly parallel tasks require no synchronization because there are no dependencies. This type of parallel programming task occurs in problems like the Black-Scholes model for stock option pricing [38]. Level two exhibits an embarrassingly parallel programming task in which all threads perform the same task, and level three requires the threads to perform a slightly different task. In level three, Alice and Bob have to move a different number of steps to reach the goal.

Moving on from embarrassingly parallel problems and into the coordination category, level four introduces the concept of exclusive ownership, which is what locks provide in parallel programs. In level four, one of the threads must acquire a key prior to reaching the goal. Only a single thread can acquire the key at a time.

The remaining problems focus on pipeline parallelism, which is used in problems such as data compression and content similarity searching [38]. In all of these levels, a wall prevents the two sprites from traversing the entire board, requiring them to work together. For example, the two threads must move a key from one side to the locked chest on the other side. The sprites can pass objects between each other using a counter that allows objects to be placed upon it between the walls. Level five, shown in Figure 3(a) requires one of the sprites to acquire the key and hand it off to the other sprite over the counter. For this level, the timing between the first sprite setting down the key and the second sprite picking up the key works out naturally if the sprites do not waste any time completing the goal. In level six, the first sprite will reach the counter before the second sprite sets down the key. This behavior models a data race between the two threads - one thread must complete a task before the other thread can begin part of its task. This particular type of data race is common in parallel programs where data must be initialized before it is used [1].



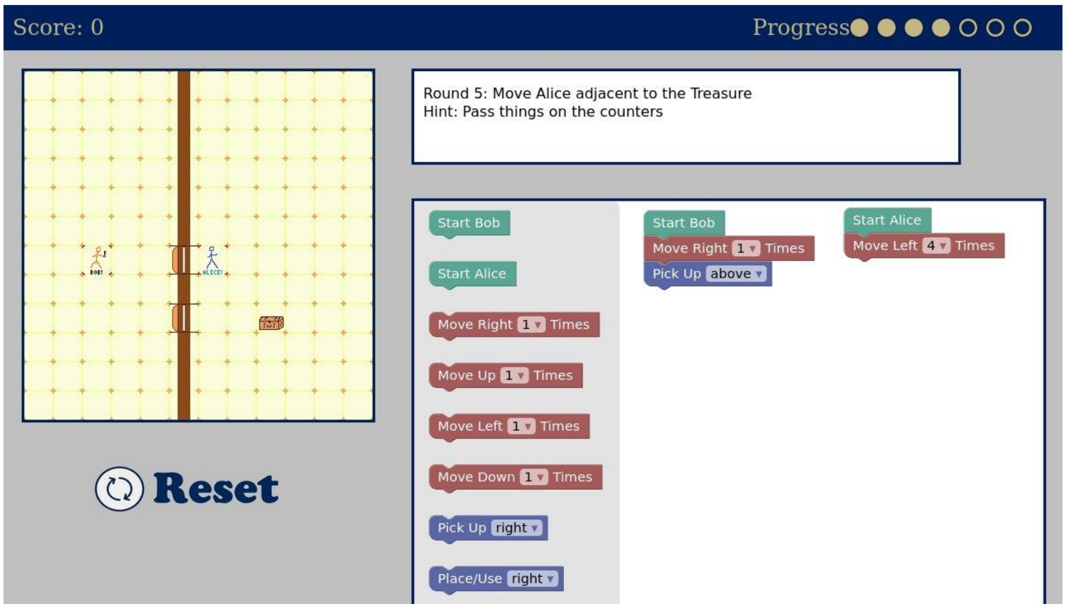
Round 5: Move Alice adjacent to the Treasure
Hint: Pass things on the counters

Start Bob
Start Alice
Move Right 1 Times
Move Up 1 Times
Move Left 1 Times
Move Down 1 Times
Pick Up right
Place/Use right

Start Bob

Start Alice

(a) Problem 5 introduces coordination between workers



Round 5: Move Alice adjacent to the Treasure
Hint: Pass things on the counters

Start Bob
Start Alice
Move Right 1 Times
Move Up 1 Times
Move Left 1 Times
Move Down 1 Times
Pick Up right
Place/Use right

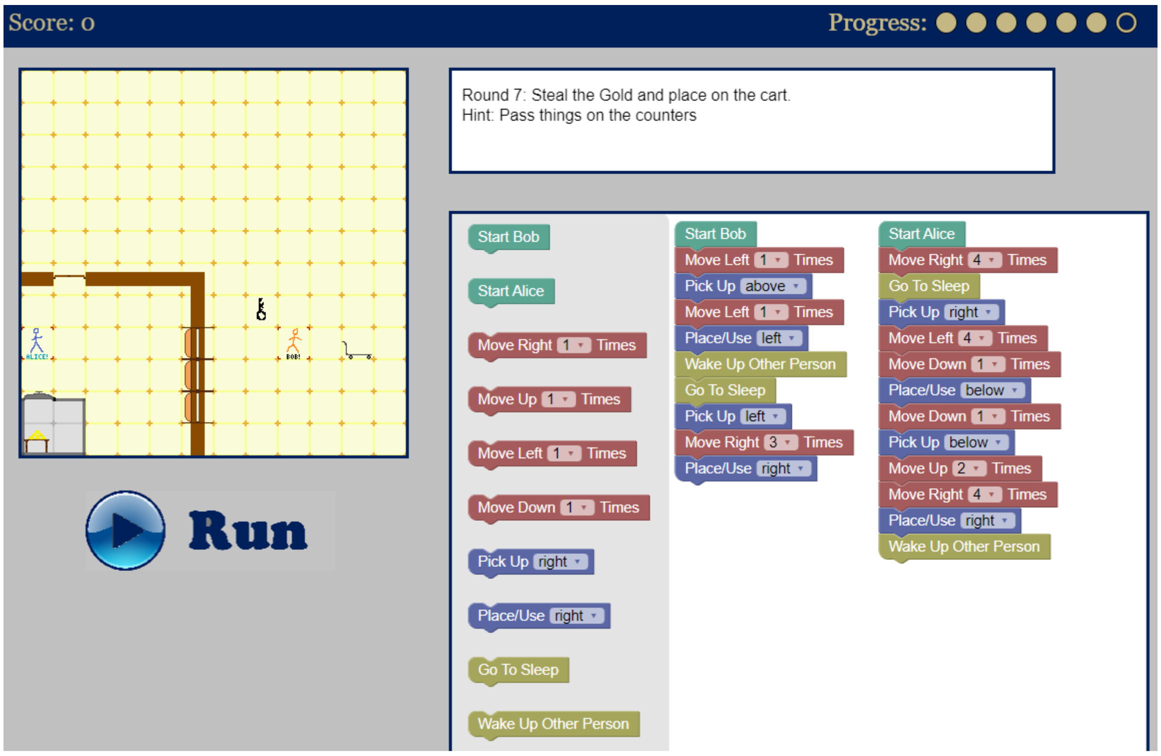
Start Bob
Move Right 1 Times
Pick Up above

Start Alice
Move Left 4 Times

(b) As the user attempts to solve the problem, the game animates their progress.

Figure 3. Problem 5 setup and in-progress screenshots.

To complete level six, the user needs to ensure that the second sprite waits for the first sprite to place the key on the counter before trying to pick it up. Users can accomplish this task by using the Go to Sleep and Wake Up Other Person blocks. When the Go to Sleep block is executed by a sprite, the sprite will do nothing until it is woken up. The Wake Up Other Person block wakes up a sprite that has been put to sleep. To achieve the optimal result for level six, Alice should Go to Sleep when she reaches the counter, and Bob should wake Alice up when he places the key on the counter. This pattern is used to solve similar problems in real-world parallel programs. Of course, part of the reason for this study is to understand if novice users will choose the same solution.


Score: 0

Progress: ● ● ● ● ● ● ○

Round 7: Steal the Gold and place on the cart.
Hint: Pass things on the counters

```
Start Bob
Start Alice
Move Right 1 Times
Move Up 1 Times
Move Left 1 Times
Move Down 1 Times
Pick Up right
Place/Use right
Go To Sleep
Wake Up Other Person
```

```
Start Bob
Move Left 1 Times
Pick Up above
Move Left 1 Times
Place/Use left
Wake Up Other Person
Go To Sleep
Pick Up left
Move Right 3 Times
Place/Use right
```

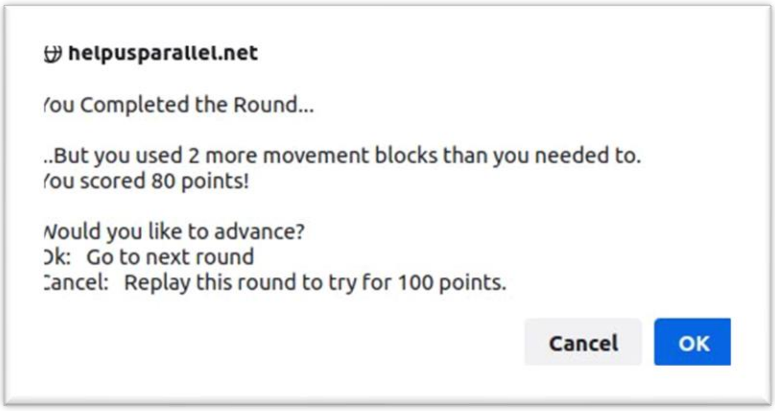
```
Start Alice
Move Right 4 Times
Go To Sleep
Pick Up right
Move Left 4 Times
Move Down 1 Times
Place/Use below
Move Down 1 Times
Pick Up below
Move Up 2 Times
Move Right 4 Times
Place/Use right
Wake Up Other Person
```

Figure 4. Problem 7 challenges novice programmers to coordinate between two workers.

The final level, shown in Figure 4, challenges the user to synchronize the sprites' actions twice and falls in the synchronization category. Bob must hand the key to Alice. Then Alice can open the vault and take the gold to Bob. Bob can then pick up the gold and place it on the cart. Completing this task requires both sprites to sleep and wake up at the appropriate times to wait for the other sprite to complete their part of the job.

3.3 Player Feedback and Scoring

After the user presses the Run button, the game window animates their solution in a step-by-step manner. Figure 3(b) demonstrates a partial solution to Problem 5 to illustrate how the sprites can move and carry objects. Prior work on serious games has identified that step-by-step program execution can be used in programming games to help the learner understand program correctness and debugging strategies [35]. If the user's code allows the sprites to complete the objective listed in the top right text box, the level is complete. If the objective has not been accomplished, the user can adjust their solution as needed and reattempt the problem as many times as needed.


helpusparallel.net

'ou Completed the Round...

..But you used 2 more movement blocks than you needed to.
'ou scored 80 points!

Nould you like to advance?
Ok: Go to next round
:ancel: Replay this round to try for 100 points.

Figure 5. After each attempt, players are scored on the number of blocks used in their solution.

Once the user has successfully solved the problem, their solution is scored based on the number of moves used compared to the number of moves used in an optimized solution to the same problem. Figure 5 shows the feedback provided to the user after completing a problem using more moves than the optimized solution. Users are scored out of 100 points on each problem, and each additional move subtracts 10 points from their total score.

4. Implementation

This framework was written in a combination of programming languages, including Javascript, HTML, CSS, PHP, and SQL. The game board, created using Javascript, is a 2D grid that represents a 2D array of memory addresses. The visual programming language was developed using Blockly version 5.2.

All data were recorded in a database using PHP and MySQL. User responses were validated using the server-side PHP script to ensure correctness and prevent malicious inputs. Responses from individual participants were collated using a combination of a browser cookie, if available, and a server-side session identifier. We tracked the time spent per attempt using both a client-side timer and a server-side timestamp.

4.1 Technical Challenges

One technical challenge is implementing a parallel game environment with a serial program. Javascript uses a single-threaded computing model. To overcome this technical challenge the game environment is simulated in parallel. The simulated parallel environment is actually serial but appears parallel by having 1 millisecond time differences between the movement of pieces. To the human eye, they seem to be moving at the same time as the human eye which can see images at 50-90 Hz and can see visual flicker up to 500 Hz [39].

The second technical challenge is synchronization between the workers. The fact that the movement of the workers is offset from each other makes it difficult to coordinate passing items to each other, detect collisions between the workers, and wake other workers. To overcome this challenge before characters are moved, the timing for every move is calculated so all moves are defined before any piece is moved or action happens. This ensures that collisions can be detected before they happen and allows movement such as one worker moving out of a position on the board while another moves into that position. This also allows coordinated actions to happen such as waking workers up so that the worker can act. Wait blocks pose an additional problem because their animation times, and all times following the wait block in the program, are dependent on the time of the corresponding wake-up block in the other worker's program. For each wait block in the code, our framework identifies the matching wake-up block in the other worker's program and resumes execution of blocks after the wait in the waiting worker's program after the time for the corresponding wake-up. If a matching wake-up block does not exist in the other worker's program, the wait block and any subsequent blocks will not happen. In the future, our framework can use this to identify and demonstrate scheduling bugs that can occur due to timing issues in the user's code [40].

5. Results

We administered this study to 95 volunteer participants drawn from students in a core course on Cyber Security. This core course is required for students in all majors that are not related to Computing. Therefore, in contrast to prior studies on parallel programming, the students participating in this study had little to no experience in both programming and parallel programming. Participants were recruited to the study via a request from their instructor and potentially an offer of extra credit in the course for completing the study.

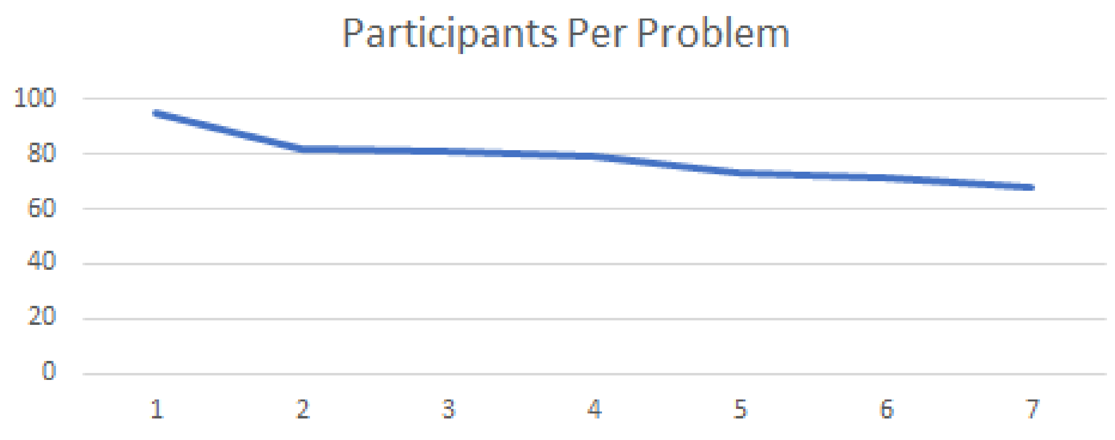


Figure 6. Number of participants completing each problem in the study

Figure 6 shows the number of participants who completed each problem in the study. 82 participants completed the second problem. This was the largest drop-off in participants throughout the study. We note that participants who failed to continue to problem 2 took less time and fewer attempts on average to complete problem 1 than participants who completed the entire study. Therefore, our best hypothesis as to why they did not continue is that they lacked interest in any reward offered for completing the study. 63 participants completed the entire study. The total number of participants and percentage of participants completing each problem in our study was similar to a prior study on using serious games to teach cybersecurity [19].

After the study was completed, we administered a brief survey to participants who were able to complete all seven problems. The survey had the student indicate their proficiency prior to the study in programming, and parallel programming, as well as comments about the process and what they found difficult. We asked them to indicate their prior programming experience from courses taught at the Naval Academy. As shown in Figure 7, prior programming experience matched what was expected from the test population considering that the population excludes students in computing majors. Table 1 describes how much prior experience participants reported in both programming and parallel programming.

The majority of participants only had experience in programming from general education courses on Cyber Security at this institution. The first course in this sequence introduces students to HTML and Javascript, and the second course introduces them to the C programming language. In total, these participants received seven weeks of instruction related to programming. Participants in this group should be familiar with how each piece of code executes sequentially. The second group of participants had some experience with programming in other courses, generally in Matlab or Python, for mathematics and statistics. Only two participants considered themselves fluent in a programming language.

Figure 7 also shows the results of a survey question on how much prior parallel programming experience participants had. The majority of participants had no prior experience with parallel programming. Twelve participants reported some knowledge of parallelism, and another four reported some experience with writing parallel code.

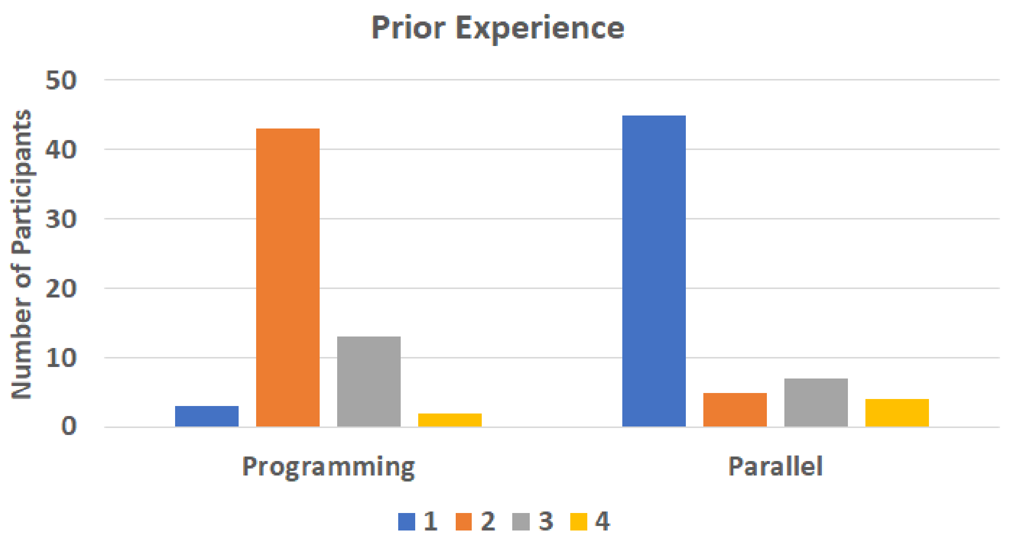


Figure 7. Prior experience in programming and parallelism prior to the study.

Table 1. Self-reported prior programming experience values.

Programming	
1	No prior experience
2	Experience understanding programs in coursework
3	Experience writing programs in coursework
4	Fluent in one programming language

Parallel	
1	No prior experience
2	Some knowledge of topics like multitasking and concurrency
3	Some knowledge of coding with threads or processes
4	Experience writing code with threads or processes

5.1 Metrics Used

When comparing the performance of groups with different levels of programming and parallel programming experience, a statistical test such as Welch's t-test may be used [41]. This test is chosen because it is appropriate for groups with unequal variance. Participants self-identified as part of the non-programmer or programmer group and as part of the non-parallel programmer or parallel programmer group. In this study, 48 participants self-identified as non-programmers and 15 participants identified as having prior programming experience. 52 participants identified as having no parallel programming experience, and 11 participants identified as having prior parallel programming experience. More detail on this breakdown is provided in Figure 7. Per-problem variances for both attempts and time per group are shown in Table 2. Typically, it is expected that experienced programmers will perform better on a given problem, requiring fewer attempts and less time to complete the task, with less variability in their performance compared to inexperienced programmers. The test results are used to determine if there is a significant difference between the groups, with a lower p-value indicating a higher likelihood of independent groups and significant differences between them. A commonly used threshold for statistical significance is a p-value less than 0.05, which suggests that any observed differences between the groups are unlikely to have occurred by chance [41]. [42, 43] use a t-test approach similar to what is presented in this work.

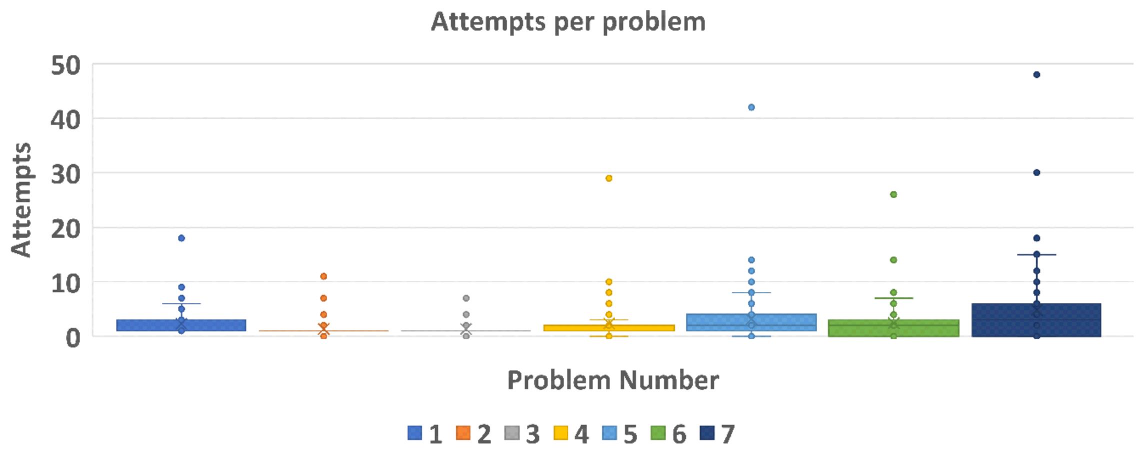


Figure 8. Number of attempts per problem

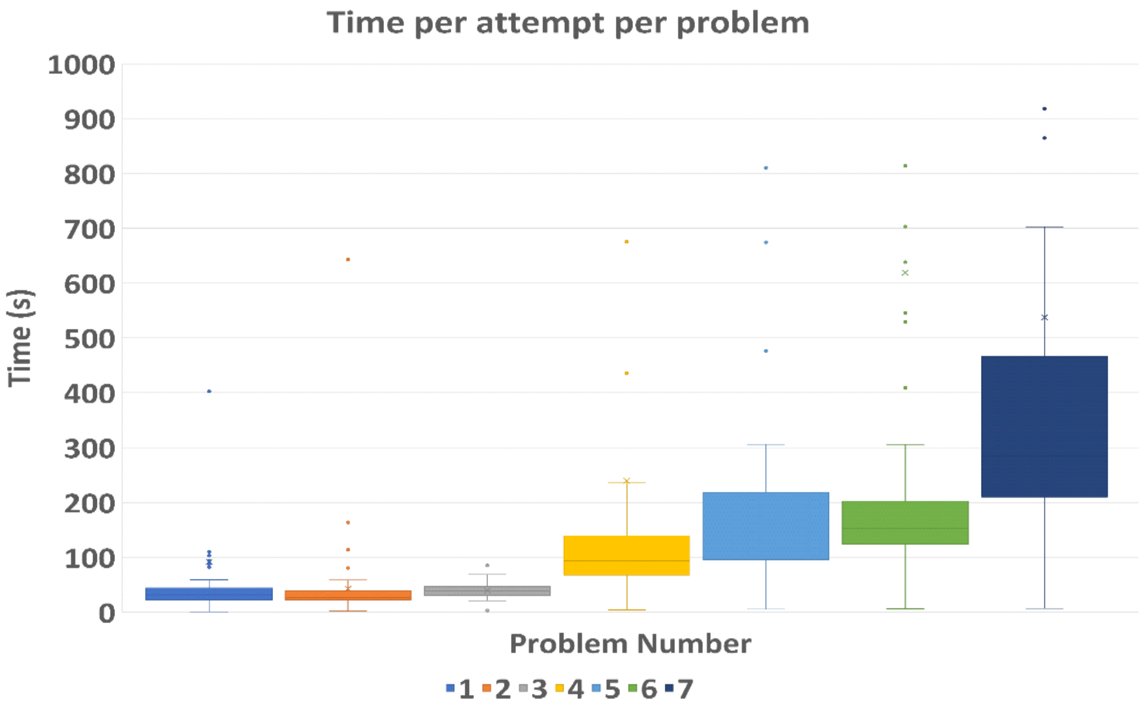


Figure 9. Average time per attempt per problem

5.2 Analysis of Student Performance

Figure 8 shows a box plot of the number of attempts taken per participant per problem. For the initial problem, participants took 2.27 attempts on average to get acquainted with the visual programming language. Likewise, Figure 9 shows a box plot of the amount of time taken per participant per problem. Participants were provided with a demonstration video that showed them how to solve the first problem, but participants were not required to watch the video before starting the study. Once they were acclimated to the visual programming language, participants required only one attempt on average to solve problems 2 and 3, which introduced parallelism with two separate sprites. Problem 4 introduced objects (a key can be seen in Figure 3(a)), and participants required 2.84 attempts on average to solve this problem due to the new mechanic. Problems 1 through 4 constituted the threading or embarrassing parallel problems as discussed in Section 3.2. Problems 5 and 6 made up the coordination category. Problem 5 introduced the concept of passing objects across a barrier that the sprites could not physically cross. Given the additional complexity of this problem, participants required 4.10 attempts on average to solve the problem. This large jump in attempts and time can be attributed to the complexity of coordination of the workers instead of the previously independent operations. Problem 6 introduced the concept of waiting for another sprite to perform some task using the "Go to sleep" and "Wake up other person" blocks. We expected this new mechanic to be a stumbling block for participants, but they solved problem 6 in 3.20 attempts on average. Finally, problem 7 (shown in Figure 4) the synchronization problem as discussed in Section 3.2, required a long sequence of blocks to solve the problem with two-way synchronization between the two sprites. This problem required more complicated logic and more blocks than every previous problem in the study. On average, participants required 6.76 attempts to solve the final problem. There were a few outliers on the last problem, with one participant requiring 48 attempts. This particular participant required above-average attempts for every problem. By studying the time per attempt and blocks changed per attempt, we could see that this participant solved the problems quickly (approximately 65 seconds per attempt) by making minor changes (2.57 blocks changed per attempt).

Table 2. Comparison of attempts and time to complete each problem for programmers vs non-programmers and parallel programmers vs non-parallel programmers.

		Programmers		Non-programmers				Parallel Programmers		Non-Parallel Programmers			
	Problem	Mean	Variance	Mean	Variance	p-value	Degrees of Freedom	Mean	Variance	Mean	Variance	p-value	Degrees of Freedom
Attempts per problem	1	1.6	0.97	2.02	2.76	0.21	40	1.73	2.22	1.96	2.41	0.62	46
	2	1.07	0.07	1.63	3.3	0.44	53	1.00	0.00	1.6	3.07	0.02	42
	3	1.27	0.35	1.39	0.66	0.58	32	1.55	0.47	1.32	0.61	0.32	44
	4	2.4	4.69	2.91	18.94	0.59	49	2.55	5.67	2.84	17.66	0.79	55
	5	2.47	1.27	4.48	40.4	0.06	55	2.00	0.8	4.42	37.27	0.01	46
	6	2.33	3.95	3.02	5.98	0.06	28	2.27	4.62	2.98	5.7	0.37	55
	7	4.2	12.03	7.43	72.87	0.04	56	4.45	6.27	7.12	70.25	0.07	53
Time in seconds per Problem	1	32.55	157.45	43.34	3,391.88	0.27	47	26.16	66.23	43.54	3,067.44	0.04	42
	2	30.55	190.55	48.51	8,328.83	0.21	53	23.91	50.57	48.53	7,682.33	0.06	42
	3	36.82	101.57	40.75	173.09	0.25	30	40.48	166.28	39.63	157.82	0.83	42
	4	101.92	3,131.62	124.16	11,004.43	0.35	41	99.73	3,659.24	123.29	10,429.77	0.36	42
	5	142.76	5,496.32	416.08	1,485,329	0.15	48	121.33	4,571.80	404.16	1,398,123	0.11	42
	6	137.31	1,010.88	844.61	9,792,969	0.14	47	139.83	1,136.26	800.74	9,232,014	0.14	42
	7	277.42	16,157.36	639.89	1,201,348	0.04	51	315.81	10,595.07	602.45	1,123,241	0.07	42

Comparing the performance of self-identified programmers versus non-programmers is useful for identifying trends in performance and what issues non-programmers encounter. For this work, we divided non-programmers from programmers by their self-reported experience survey questions outlined in Table 1, with 2 and below being non-programmers and 3 and above being programmers. Similarly, non-parallel programmers were those that self-identified as 2 or lower in prior parallel programming and parallel programmers were 3 and above as indicated in Table 1. Using individuals that completed all problems, there are 15 self-identified programmers and 48 self-identified non-programmers. In the parallel group there were 11 self-identified parallel programmers and 52 self-identified non-parallel programmers. The performance on problems 1 through 7 is shown in Table 2. This table shows the mean attempts and time for each problem along with the variance of the groups. Additionally, the p-values for the two-tailed Welch's t-test are given along with the corresponding degrees of freedom using the Welch-Satterthwaite equation [41].

We hypothesized that both the self-identified programmers and parallel programmers would perform better than their non-programmer counterparts by needing fewer attempts and less time for each problem. As shown in Table 2, in all cases the mean number of attempts and time to solve the problem were smaller in the proficient group and the variance was lower. Problems 1 through 4, the embarrassing parallel problems, the programmers and nonprogrammers the groups were not separable based on their p-values. This is understandable as the beginning problems are designed to introduce the game. Examining problems 5 through 7, which test parallel programming concepts, a larger separation comes into focus culminating in problem 7, the synchronization problem. Similar results are seen in the non-programmers vs parallel programmers. The number of attempts and time correspond to the non-programming group tending to do more of a incremental build and test approach, where you make a piece of code, ensure that it works, and then iteratively repeat this until you have the full solution. The groups with more experience took fewer attempts so it is more difficult to draw out the approach they took. In the end, the groups tended to get a similar average number of lines in the solution. The lines of code in the final submission for both programmers and nonprogrammers were similar to each other, showing that non-programmers were able to obtain a efficient solution albeit needing more time and attempts to obtain this.

5.3 Learning Outcome Analysis

We evaluated the learning outcomes of participants using a similar methodology to Serrano-Laguna et al [44]. For future studies on programming frameworks, this methodology should serve as a guide for how to assess the usability of a framework or compare multiple competing frameworks.

For the initial assessment, we evaluate participants on problem 5 because problem 5 is the last problem to introduce a new concept. For the final assessment, we evaluate participants on problem 7, which is the final and most difficult problem. We score participants based on the total number of attempts to complete the problem, on the time taken to complete the problem, and on how optimal their solution is, as described in Section 3.3. For each of these parameters, we scale the raw data to a factor between 0 and 1. For attempts and time, we subtract that factor from 1 so that taking fewer attempts or less time produces a higher value. For example, two participants, A and B, solved a problem in 2 and 6 attempts, respectively. We scale by the 90th percentile instead of the maximum to avoid scaling by an outlier. The 90th percentile number of attempts for this problem was 12. By subtracting the minimum number of attempts (1) and dividing by the 90th percentile number of attempts, we get scaled(A) = (2 − 1)/(12 − 1) = 0.09 and scaled(B) = (6 − 1)/(12 − 1) = 0.45. We subtract these scaled factors from 1 so that higher is better, yielding weight(A) = 0.91 and weight(B) = 0.55. We note that for outliers O such that scaled(O) > 1, we set scaled(O) = 1 and therefore weight(O) = 0 so that a single outlier score will not disproportionately affect a participant's aggregate score. We aggregate these three factors using equal weights to produce a single number representing how well the participant performed on that problem. An aggregate score of 3 would indicate that the participant solved the problem in the least number of attempts, in the fastest time, and produced the most optimal possible solution.

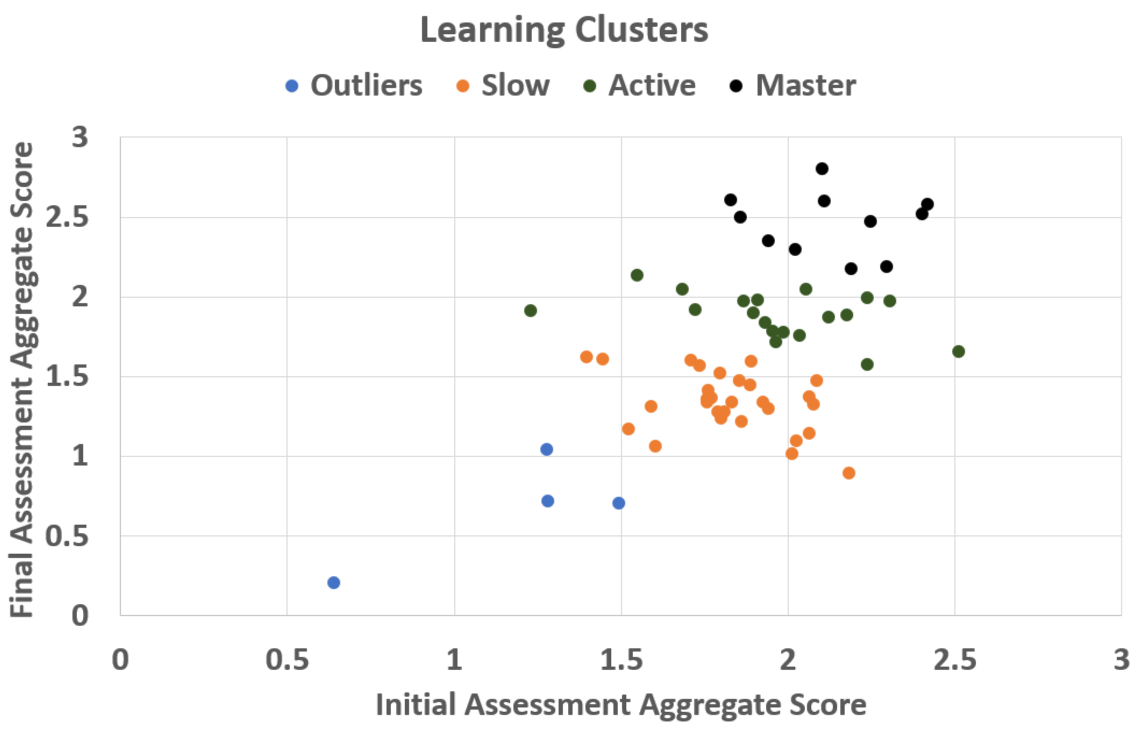


Figure 10. Kmeans classification of learning throughout the study

Figure 10 shows the results of this analysis. We applied a KMeans clustering algorithm with 4 means to identify groups of participants. 11 participants were classified as masters because they achieved high scores on both the initial and final assessments. These participants demonstrated strong programming abilities throughout the study and would have likely performed well given any reasonable programming framework. 4 outliers struggled to solve the programming problems throughout the entire study. In the middle, 29 participants were classified as slow learners, and 19 participants were classified as fast learners. From Figure 10, we can see that fast learners generally performed better than slow learners on the final assessment despite similar scores on the initial assessment.

We analyzed the factors in the aggregate score based on the learning classification. Figure 11 shows the average time per problem per learning class. Slow learners generally took more time to solve the first 6 problems. However, the additional time spent on the first 6 problems seems to have paid off for the slow learners on problem 7 as they outperformed the fast learners. From Figure 12, we can see that the Slow learners generally required slightly more attempts on average to solve each problem, but the slow and fast learners required a similar number of attempts on problem 7. Figure 13 shows the average score, as defined in Section 3.3, for each problem per learning class. The most striking difference in score occurs on problem 7, where the slow and fast learners take 3-4 additional and unnecessary steps to solve the problem compared to master learners.

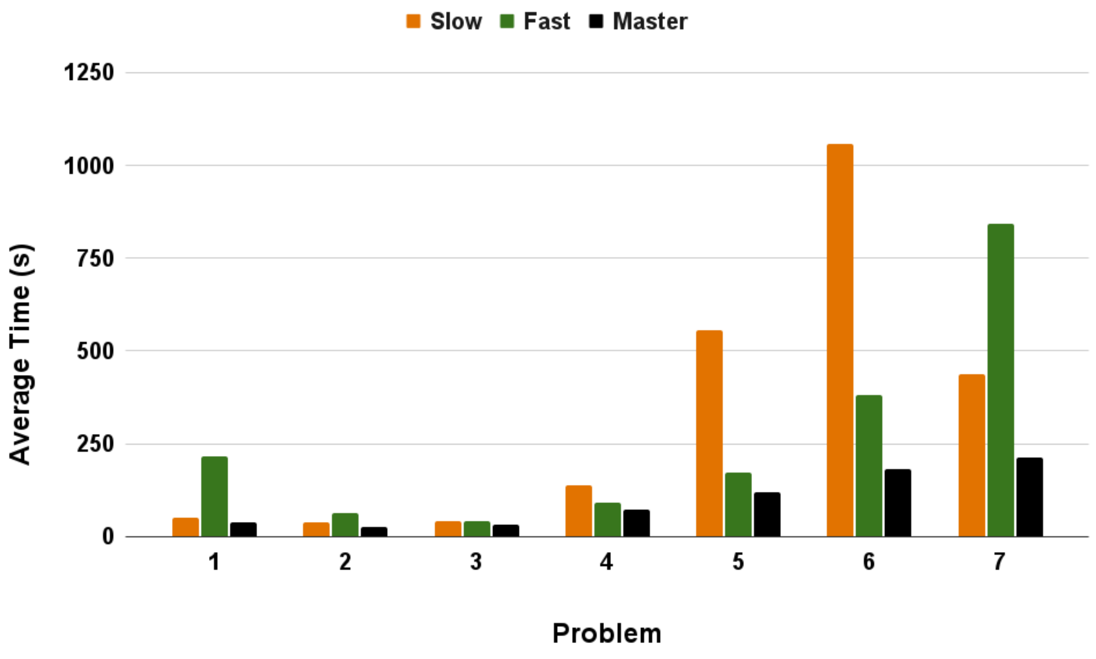


Figure 11. Time per problem grouped by kmeans classification of learning outcomes

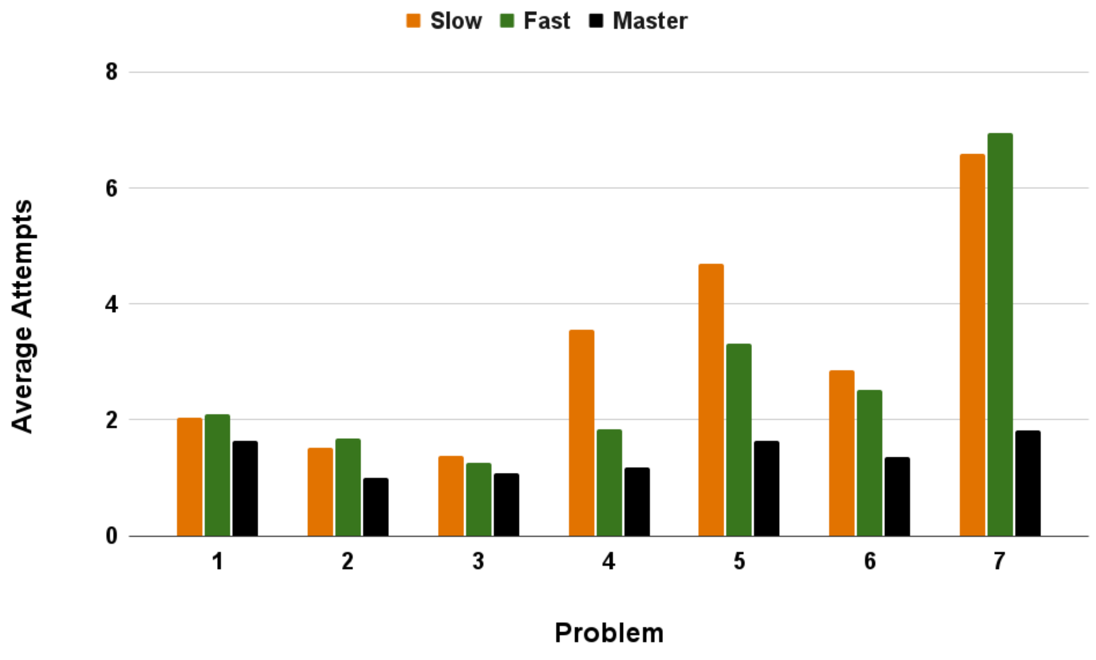


Figure 12. Attempts per problem grouped by kmeans classification of learning outcomes

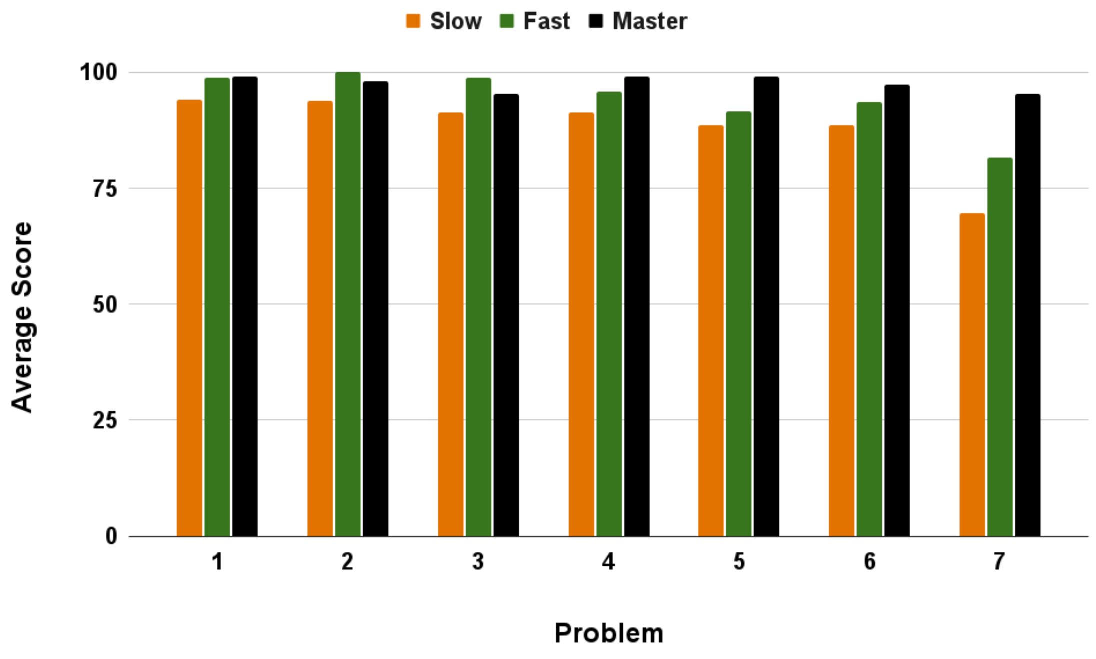


Figure 13. Score per problem grouped by kmeans classification of learning outcomes

To compare two or more programming methodologies, we recommend that a common set of problems are used for both the initial and final assessments. Learning clusters can be compared to assess the usability of each programming methodology. Time and attempts taken to solve a problem can be used to assess how usable a programming methodology is by novice programmers, and scores for participant solutions can be used to compare how effective novice programmers are at finding optimized solutions to each problem. Further work will be required to demonstrate the use of this framework in a comparative study.

5.4 Qualitative Participant Feedback

At the end of study, we asked participants "What challenges did you encounter while completing this study?" Participants generally responded with a self-assessment of the mistakes that they made while playing the game. Of the participants who completed the study, 43 participants (65%) responded to this question. We share common themes in the feedback.

The most common response was that participants were confused as to how the game environment worked and therefore had to experiment to understand how to solve the problems. For example, one participant responded that they "were not entirely sure which part of the wall was a counter or if the entire wall would act like a counter." In future studies using this framework, we hope to keep participants from being confused by game mechanics by using better-animated sprite libraries that illustrate the purpose of each object in the game. Other participants noted that they counted the number of moves incorrectly while attempting to solve the problems, and some indicated that they took multiple attempts to "try to figure the least amount of moves." A few participants were challenged by the timing of coordination between the two sprites and took additional attempts to "figure out which order to put the actions in." We plan to take these challenges into account as we design future studies using this framework.

5.5 Limitations

This article focuses on the design of a visual programming language and serious game for assessing novice programmers’ ability to write correct parallel programs. This framework has not been used to compare the use of two or more parallel programming languages or libraries by novice programmers. Further work is required to determine the best practices for comparing the usability of programming systems with serious games.

Participation in this study was conducted remotely, and the method for timing participants on each problem could not account for idle time on the website. Therefore, outliers exist in the timing data, likely due to students multitasking while participating in the study.

Problems were scored based on the number of animated steps to complete the problem, which may not be a good indicator of program performance. For more complicated parallel programming problems, a better solution would be to generate code in a language like C++ or Java from the block program and collect performance data on a real computer system.

6. Future Work and Conclusions

In this paper, we have demonstrated the use of visual programming languages and serious games as a framework for studying novice programmers' abilities to use complex programming features. This is a novel use of serious games that could be applied to research on new programming languages and methodologies. This work fills a gap in the limited usability studies performed on new programming languages and methodologies by demonstrating the effectiveness of using visual programming languages and serious games to perform usability studies using novice programmers. This approach can be applied to research on new programming languages and methodologies.

Several parts of this work can be generalized to other studies in programming education and research. First, the use of Blockly and serious games to represent complex programming language concepts can be used to analyze future work that proposes new programming languages and methodologies. Second, the analysis performed in the results section can be used by researchers to analyze and compare programming languages and methodologies, specifically as they apply to novice programmers. Finally, the insights learned from performing this study can be used to improve future studies on programming languages and methodologies using visual programming languages and serious games.

This work provides a framework for additional studies. The ability to understand how novice programmers approach a complex problem can guide how different parallel frameworks can be implemented to appeal to novice programmers. Additionally, the ability to understand how novice programmers think can lead to breakthroughs in teaching parallel concepts. The framework presented here can be expanded upon to different implementations to include locks, mutexes, and semaphores and each can be investigated for its ease of implementation. We plan to conduct a more thorough study of how well novice programmers can understand existing parallel programming paradigms and use this framework to conduct comparative studies of parallel programming methodologies.

From the initial study we conducted using this framework, we can conclude that novice programmers can solve complex programming problems using a visual programming language paired with a serious game. Participants in the study were largely novices in both programming and parallel programming, but they were able to solve parallel programming problems using the visual programming language. Participants with more prior experience in programming took less attempts to solve these problems, but those with less prior experience were still able to produce a working solution. We hope that this framework can be applied to other programming languages, concepts, and methodologies to evaluate the usability of features using novice programmers.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback and suggestions for improving this paper. Thank you to ENS Matthew Friedel for creating the prototype Blockly language and game design. Finally, we would like to thank the instructors in the Department of Electrical and Computer Engineering at the United States Naval Academy for recruiting participants for the user study and the students who participated in this study.

Conflicts of interest

We have no conflicts of interest to declare for this work.

References

[1] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: A comprehensive study on real world concurrency bug characteristics,” in Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS XIII. New York, NY, USA: Association for Computing Machinery, 2008, doi: 10.1145/1346281.1346323.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” in Proceedings of the 44th Annual International Symposium on Computer Architecture, ser. ISCA ’17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3079856.3080246.

[3] The importance of domain knowledge. https://blog.ml.cmu.edu/2020/08/31/1-domain-knowledge/. Accessed: 2022-07-06.

[4] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz, “Parallel programmer productivity: A case study of novice parallel programmers,” in SC ’05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, 2005. doi: https://doi.org/10.1109/SC.2005.53.

[5] V. Pankratius and A.-R. Adl-Tabatabai, “A study of transactional memory vs. locks in practice,” in Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures, ser. SPAA ’11. New York, NY, USA: Association for Computing Machinery, 2011. doi: 10.1145/1989493.1989500.

[6] S. Timur et al, "PRE-SERVICE PRE-SCHOOL TEACHERS' OPINIONS ABOUT USING BLOCK-BASED CODING/SCRATCH," Acta Didactica Napocensia, vol. 14, (2), pp. 299-317, 2021. Available: https://www.proquest.com/scholarly-journals/pre-service-school-teachers-opinions-about-using/docview/2623909591/se-2. doi: https://doi.org/10.24193/adn.14.2.22.

[7] S. Papadakis, “The impact of coding apps to support young children in computational thinking and computational fluency. a literature review,” Frontiers in Education, vol. 6, 2021. doi: 10.3389/feduc.2021.657895.

[8] D. Weintrop and U. Wilensky, “Comparing block-based and text-based programming in high school computer science classrooms,” ACM Trans. Comput. Educ., vol. 18, no. 1, oct 2017. doi: 10.1145/3089799.

[9] T. Urness and E. Manley, “Building a thriving cs program at a small liberal arts college,” J. Comput. Sci. Coll., vol. 26, no. 5, pp. 268–274, may 2011, doi: 10.5555/1961574.1961630.

[10] Hour of code. https://hourofcode.com/. Accessed: 2022-04-20.

[11] K. Brennan and M. Resnick, “New frameworks for studying and assessing the development of computational thinking,” in Proceedings of the 2012 annual meeting of the American Educational Research Association (AERA), 2012. Retrieved from: https://web.media.mit.edu/~kbrennan/files/Brennan_Resnick_AERA2012_CT.pdf

[12] D. Weintrop and U. Wilensky, “How block-based, text-based, and hybrid block/text modalities shape novice programming practices,” International Journal of Child-Computer Interaction, vol. 17, pp. 83–92, 2018. doi: https://doi.org/10.1016/j.ijcci.2018.04.005.

[13] R. Kraleva, V. Kralev, and D. Kostadinova, “A methodology for the analysis of blockbased programming languages appropriate for children,” vol. JCSE, Volume 13, pp. pp.1–10, 03 2019. doi: 10.5626/JCSE.2019.13.1.1.

[14] João, Nuno, Fábio, and Ana, “A Cross-analysis of Block-based and Visual Programming Apps with Computer Science Student-Teachers,” Education Sciences, vol. 9, no. 3, p. 181, Jul. 2019, doi: 10.3390/educsci9030181.

[15] D. Weintrop, “Block-based programming in computer science education,” Commun. ACM, vol. 62, no. 8, pp. 22–25, jul 2019. doi: 10.1145/3341221.

[16] E. Macrides, O. Miliou, and C. Angeli, “Programming in early childhood education: A systematic review,” International Journal of Child-Computer Interaction, vol. 32, p. 100396, 2022. doi: https://doi.org/10.1016/j.ijcci.2021.100396.

[17] A. Wilson and D. Moffatt, “Evaluating scratch to introduce younger schoolchildren to programming,” Jan. 2010, Paper presented at the 22nd AnnualWorkshop of the Psychology of Programming Interest Group, Leganés, Spain, 19-21 September 2010. Retrieved from: https://ppig.org/files/2010-PPIG-22nd-Wilson.pdf.

[18] N. Zaric, V. Lukarov, and U. Schroder, “A fundamental study for gamification design: Exploring learning tendencies & effects,” International Journal of Serious Games, vol. 7, no. 4, pp. 3–25, Dec. 2020. doi: 10.17083/ijsg.v7i4.356.

[19] M. Katsantonis and I. Mavridis, “Evaluation of hacklearn cofelet game user experience for cybersecurity education,” International Journal of Serious Games, vol. 8, no. 3, pp. 3–24, Sep. 2021. doi: 10.17083/ijsg.v8i3.437.

[20] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. De Halleux, “Code hunt: Experience with coding contests at scale,” in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, 2015. doi: 10.1109/ICSE.2015.172

[21] A. J. Abdellatif, B. McCollum, and P. McMullan, “Serious games: Quality characteristics evaluation framework and case study,” in 2018 IEEE Integrated STEM Education Conference (ISEC), 2018. doi: 10.1109/ISECon.2018.8340460.

[22] N. Zaric, R. Roepke, V. Lukarov, and U. Schroeder, “Gamified learning theory: The moderating role of learners & learning tendencies,” International Journal of Serious Games, vol. 8, no. 3, pp. 71–91, Sep. 2021. doi: 10.17083/ijsg.v8i3.438.

[23] M. A. Miljanovic and J. S. Bradbury, “Making serious programming games adaptive,” in Serious Games, S. Göbel, A. Garcia-Agundez, T. Tregel, M. Ma, J. Baalsrud Hauge, M. Oliveira, T. Marsh, and P. Caserman, Eds. Cham: Springer International Publishing, 2018. ISBN 978-3-030-02762-9 pp. 253–259, doi: 10.1007/978-3-030-02762-9_27.

[24] G. Gris and C. Bengtson, “Assessment measures in game-based learning research: A systematic review,” International Journal of Serious Games, vol. 8, no. 1, pp. 3–26, Mar. 2021. doi: 10.17083/ijsg.v8i1.383.

[25] A. Calderón and M. Ruiz, “A systematic literature review on serious games evaluation: An application to software project management,” Computers & Education, vol. 87, pp. 396–422, Sep. 2015, doi: https://doi.org/10.1016/j.compedu.2015.07.011.

[26] P. Moreno-Ger, J. Torrente, Y. G. Hsieh, and W. T. Lester, “Usability testing for serious games: Making informed design decisions with user data,” Adv. in Hum.-Comp. Int., vol. 2012, jan 2012. doi: 10.1155/2012/369637.

[27] K. Mitgutsch and N. Alvarado, “Purposeful by design? a serious game design assessment framework,” in Proceedings of the International Conference on the Foundations of Digital Games, ser. FDG ’12. New York, NY, USA: Association for Computing Machinery, 2012. doi: 10.1145/2282338.2282364.

[28] I. Ghergulescu and C. H. Muntean, Measurement and Analysis of Learner's Motivation in Game-Based E-Learning. New York, NY: Springer New York, 2012, pp. 355–378, doi: https://doi.org/10.1007/.
[29] A. All, E. P. Nunez Castellar, and J. Van Looy, "Measuring effectiveness in digital game-based learning: A methodological review." International Journal of Serious Games, vol. 1, no. 2, Jun. 2014. doi: 10.17083/ijsg.v1i2.18.
[30] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol. 21, no. 7, pp. 558–565, jul 1978. doi: 10.1145/359545.359563.
[31] L. Hochstein, V. R. Basili, M. V. Zelkowitz, J. K. Hollingsworth, and J. Carver, "Combining self-reported and automatic data to improve programming effort measurement," ser. ESEC/FSE-13. New York, NY, USA: Association for Computing Machinery, 2005. doi: 10.1145/1081706.1081762.
[32] C. DeLozier, A. Eizenberg, B. Lucia, and J. Devietti, "Sofritas: Serializable orderingfree regions for increasing thread atomicity scalably," in Proceedings of the Twenty- Third International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018. doi: 10.1145/3173162.3173192.
[33] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, Parallel programming in OpenMP. Morgan kaufmann, 2001, doi: https://dl.acm.org/doi/10.5555/355074.
[34] M. P. Forum, "Mpi: A message-passing interface standard," USA, Tech. Rep., 1994, doi: https://dl.acm.org/doi/10.5555/898758.
[35] A. Areizaga Blanco and H. Engstraum, "Patterns in mainstream programming games," International Journal of Serious Games, vol. 7, no. 1, pp. 97–126, Mar. 2020. doi: 10.17083/ijsg.v7i1.335.
[36] Google. Blockly. https://developers.google.com/blockly. Accessed: 2022-04-20.
[37] T. Mattson, "Scientific computation," in Parallel and Distributed Computing Handbook, A. Zomaya, Ed. New York, NY: McGraw-Hill, 1996, ch. 34, pp. 981–1002, doi: https://dl.acm.org/doi/10.5555/230865.
[38] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011, doi: https://dl.acm.org/doi/10.5555/2125903.
[39] J. Davis, Y. Hsieh, and H. Lee, "Humans perceive flicker artifacts at 500 hz," Sci. Rep, Feb 2015. doi: 10.1038/srep07861.
[40] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS XV. New York, NY, USA: Association for Computing Machinery, 2010. doi: 10.1145/1736020.1736040.
[41] B. L. Welch, "The generalization of 'student's' problem when several different population varlances are involved," Biometrika, vol. 34, no. 1-2, pp. 28–35, 01 1947. doi: 10.1093/biomet/34.1-2.28.
[42] W. Westera, "Comparing bayesian statistics and frequentist statistics in serious games research," International Journal of Serious Games, vol. 8, no. 1, pp. 27–44, Mar. 2021. doi: 10.17083/ijsg.v8i1.403.
[43] R. Roepke, V. Drury, U. Meyer, and U. Schroeder, "Exploring and evaluating different game mechanics for anti-phishing learning games," International Journal of Serious Games, vol. 9, no. 3, pp. 23–41, Sep. 2022. doi: 10.17083/ijsg.v9i3.501.
[44] M. B. F. M. F. M. B. Serrano-Laguna, A., "A methodology for assessing the effectiveness of serious games and for inferring player learning outcomes." pp. 2849–2871. doi: https://doi.org/10.1007/s11042-017-4467-6.